33 | 仓库划分：文件系统的格式化操作

2021-07-23 LMOS

《操作系统实战45讲》 课程介绍 >

讲述：陈晨

时长 14:09 大小 12.97M

你好，我是 LMOS。

上一节课中，我们已经设计好了文件系统数据结构，相当于建好了仓库的基本结构。

今天，我将和你一起探索仓库的划分，即什么地方存放仓库的管理信息，什么地方存放进程的“劳动成果”（也就是文件），对应于文件系统就是文件系统的格式化操作。

具体我是这样安排的，我们先来实现文件系统设备驱动，接着建立文件系统超级块，然后建立根目录，最后建立文件系统的位图。下面，我们先从建立文件系统设备开始。

这节课的配套代码，你可以从这里获取。

文件系统设备

根据我们前面的设计，文件系统并不是 Cosmos 的一部分，它只是 Cosmos 下的一个设备。

既然是设备，那就要编写相应的设备驱动程序。我们首先得编写文件系统设备的驱动程序。由于前面已经写过驱动程序了，你应该对驱动程序框架已经很熟悉了。

我们先在 cosmos/drivers/ 目录下建立一个 drvrfs.c 文件，在里面写下文件系统驱动程序框架代码，如下所示。

复制代码

```
drvstus_t rfs_entry(driver_t* drvp,uint_t val,void* p){……}
drvstus_t rfs_exit(driver_t* drvp,uint_t val,void* p){……}
drvstus_t rfs_open(device_t* devp,void* iopack){……}
drvstus_t rfs_close(device_t* devp,void* iopack){……}
drvstus_t rfs_read(device_t* devp,void* iopack){……}
drvstus_t rfs_write(device_t* devp,void* iopack){……}
drvstus_t rfs_lseek(device_t* devp,void* iopack){……}
drvstus_t rfs_ioctrl(device_t* devp,void* iopack){……}
drvstus_t rfs_dev_start(device_t* devp,void* iopack){……}
drvstus_t rfs_dev_stop(device_t* devp,void* iopack){……}
drvstus_t rfs_set_powerstus(device_t* devp,void* iopack){……}
drvstus_t rfs_enum_dev(device_t* devp,void* iopack){……}
drvstus_t rfs_flush(device_t* devp,void* iopack){……}
drvstus_t rfs_shutdown(device_t* devp,void* iopack){……}
```

这个框架代码我们已经写好了，是不是感觉特别熟悉？这就是我们开发驱动程序的规范操作。下面，我们来建立文件系统设备。

按照之前的设计（如果不熟悉可以回顾第 32 课），我们将使用 4MB 内存空间来模拟真实的储存设备，在建立文件系统设备的时候分配一块 4MB 大小的内存空间，这个内存空间我们用一个数据结构来描述，这个数据结构的分配内存空间的代码如下所示。

复制代码

```
typedef struct s_RFSDEVEXT
{
    spinlock_t rde_lock;//自旋锁
    list_h_t rde_list;//链表
    uint_t rde_flg;//标志
```

```
    uint_t rde_stus;//状态
    void* rde_mstart;//用于模拟储存介质的内存块的开始地址
    size_t rde_msize;//内存块的大小
    void* rde_ext;//扩展所用
}rfsdevext_t;
drvstus_t new_rfsdevext_mmblk(device_t* devp,size_t blksz)
{
    //分配模拟储存介质的内存空间，大小为4MB
    adr_t blkp= krlnew(blksz);
    //分配rfsdevext_t结构实例的内存空间
    rfsdevext_t* rfsexp=(rfsdevext_t*)krlnew(sizeof(rfsdevext_t));
    //初始化rfsdevext_t结构
    rfsdevext_t_init(rfsexp);
    rfsexp->rde_mstart=(void*)blkp;
    rfsexp->rde_msize=blksz;
    //把rfsdevext_t结构的地址放入device_t 结构的dev_extdata字段中，这里dev_extdata字
    devp->dev_extdata=(void*)rfsexp;.
    return DFCOKSTUS;
}
```

上述代码中，new_rfsdevext_mmblk 函数分配了一个内存空间和一个 rfsdevext_t 结构实例变量，rfsdevext_t 结构中保存了内存空间的地址和大小。而 rfsdevext_t 结构的地址放在了 device_t 结构的 dev_extdata 字段中。

剩下的就是建立文件系统设备了，我们在文件系统驱动程序的 rfs_entry 函数中，通过后面这段代码完成这个功能。

复制代码

```
void rfs_set_device(device_t* devp,driver_t* drvp)
{
    //设备类型为文件系统类型
    devp->dev_id.dev_mtype = FILESYS_DEVICE;
    devp->dev_id.dev_stype = 0;
    devp->dev_id.dev_nr = 0;
    //设备名称为rfs
    devp->dev_name = "rfs";
    return;
}
drvstus_t rfs_entry(driver_t* drvp,uint_t val,void* p)
{
    //分配device_t结构并对其进行初级初始化
    device_t* devp = new_device_dsc();
    rfs_set_driver(drvp);
    rfs_set_device(devp,drvp);
    //分配模拟储存设备的内存空间
    if(new_rfsdevext_mmblk(devp,FSMM_BLK) == DFCERRSTUS){……}
    //把设备加入到驱动程序之中
```

```
    if(krldev_add_driver(devp,drvp) == DFCERRSTUS){……}
    //向内核注册设备
    if(krlnew_device(devp)==DFCERRSTUS){……}
    return DFCOKSTUS;
}
```

其实这和我们之前的写 systick 驱动程序的套路差不多，只不过这里需要分配一个模拟储存设备的空间，并把它放在 device_t 结构相关的字段中。**还有很重要的一点是，这个设备类型我们要在 rfs_set_device 函数把它设置好，设置成文件系统类型。**

需要注意的是要把 rfs_entry 函数放在驱动表中，文件系统程序才可以运行，下面我们就把这个 rfs_entry 函数，放入驱动表中，代码如下所示。

复制代码

```
//cosmos/kernel/krlglobal.c
KRL_DEFGLOB_VARIABLE(drventyexit_t,osdrvetytabl)[]={systick_entry,rfs_entry,NU
```

有了上述代码，Cosmos 在启动的时候，在 init_krldriver 函数中就会运行 rfs_entry 函数。从名字就能看出 rfs_entry 函数的功能，这是 rfs 文件系统设备驱动程序的入口函数，它一旦执行，就会建立文件系统设备。

文件系统系统格式化

我们经常听说格式化硬盘、格式化 U 盘，可以把设备上的数据全部清空，事实是格式化操作并不是把设备上所有的空间都清零，而是在这个设备上重建了文件系统用于管理文件的那一整套数据结构。这也解释了为什么格式化后的设备，还能通过一些反删除软件找回一些文件。

在储存设备上创建文件系统，其实就是执行这个格式化操作，即重建文件系统的数据结构。

那么接下来，我们就从建立文件系统的超级块开始，然后建立用于管理储存设备空间的位图，最后建立根目录，这样才能最终实现在储存设备上创建文件系统。

建立超级块

我们首先来建立文件系统的超级块。建立超级块其实非常简单，就是初始化超级块的数据结构，然后把它写入到储存设备中的第一块逻辑储存块。

下面我们一起写代码来实现，如下所示。

复制代码

```
void *new_buf(size_t bufsz)
{
    return (void *)krlnew(bufsz);//分配缓冲区
}
void del_buf(void *buf, size_t bufsz)
{
    krldelete((adr_t)buf, bufsz)//释放缓冲区
    return;
}
void rfssublk_t_init(rfssublk_t* initp)
{
    krlspinlock_init(&initp->rsb_lock);
    initp->rsb_mgic = 0x142422;//标志就是一个数字而已，无其它意义
    initp->rsb_vec = 1;//文件系统版本为1
    initp->rsb_flg = 0;
    initp->rsb_stus = 0;
    initp->rsb_sz = sizeof(rfssublk_t);//超级块本身的大小
    initp->rsb_sblksz = 1;//超级块占用多少个逻辑储存块
    initp->rsb_dblksz = FSYS_ALCBLKSZ;//逻辑储存块的大小为4KB
    //位图块从第1个逻辑储存块开始，超级块占用第0个逻辑储存块
    initp->rsb_bmpbks = 1;
    initp->rsb_bmpbknr = 0;
    initp->rsb_fsysallblk = 0;
    rfsdir_t_init(&initp->rsb_rootdir);//初始化根目录
    return;
}
bool_t create_superblk(device_t *devp)
{
    void *buf = new_buf(FSYS_ALCBLKSZ);//分配4KB大小的缓冲区，清零
    hal_memset(buf, 0, FSYS_ALCBLKSZ);
    //使rfssublk_t结构的指针指向缓冲区并进行初始化
    rfssublk_t *sbp = (rfssublk_t *)buf;
    rfssublk_t_init(sbp);
    //获取储存设备的逻辑储存块数并保存到超级块中
    sbp->rsb_fsysallblk = ret_rfsdevmaxblknr(devp);
    //把缓冲区中超级块的数据写入到储存设备的第0个逻辑储存块中
    if (write_rfsdevblk(devp, buf, 0) == DFCERRSTUS)
    {
        return FALSE;
    }
    del_buf(buf, FSYS_ALCBLKSZ);//释放缓冲区
    return TRUE;
}
```

上述代码的意思是，我们先在内存缓冲区中建立文件系统的超级块，最后会调用 write_rfsdevblk 函数，把内存缓冲区的数据写入到储存设备中。

下面我们来实现这个 write_rfsdevblk 函数，代码如下所示。

复制代码

```
//返回设备扩展数据结构
rfsdevext_t* ret_rfsdevext(device_t* devp)
{
    return (rfsdevext_t*)devp->dev_extdata;
}
//根据块号返回储存设备的块地址
void* ret_rfsdevblk(device_t* devp,uint_t blknr)
{
    rfsdevext_t* rfsexp = ret_rfsdevext(devp);
    //块号乘于块大小的结果再加上开始地址（用于模拟储存设备的内存空间的开始地址）
    void* blkp = rfsexp->rde_mstart + (blknr*FSYS_ALCBLKSZ);
    //如果该地址没有落在储存入设备的空间中，就返回NULL表示出错
    if(blkp >= (void*)((size_t)rfsexp->rde_mstart+rfsexp->rde_msize))
        return NULL;
    //返回块地址
    return blkp;
}
//把4KB大小的缓冲区中的内容，写入到储存设备的某个逻辑储存块中
drvstus_t write_rfsdevblk(device_t* devp,void* weadr,uint_t blknr)
{
    //返回储存设备中第blknr块的逻辑存储块的地址
    void* p = ret_rfsdevblk(devp,blknr);
    //复制数据到逻辑储存块中
    hal_memcpy(weadr,p,FSYS_ALCBLKSZ);
    return DFCOKSTUS;
}
```

前面我们一下子写了三个函数，由于我们用内存模拟储存设备，我们要写一个 ret_rfsdevext 函数返回设备扩展数据结构，这个函数和 ret_rfsdevblk 函数将会一起根据块号，计算出内存地址。然后，我们把缓冲区的内容复制到这个地址开始的内存空间就行了。

建立位图

接下来，我们要建立文件系统的位图了。

延续我们文件系统的设计思路，储存设备被分成了许多同等大小的逻辑储存块，位图就是为了能准确地知道储存设备中，哪些逻辑储存块空闲、哪些是被占用的。

我们使用一个逻辑储存块空间中的所有字节，来管理逻辑储存块的状态。建立位图无非就是把储存设备中的位图块清零，因为开始文件系统刚创建时，所有的逻辑储存块都是空闲的。下面我们来写好代码。

复制代码

```
//把逻辑储存块中的数据，读取到4KB大小的缓冲区中
drvstus_t read_rfsdevblk(device_t* devp,void* rdadr,uint_t blknr)
{
    //获取逻辑储存块地址
    void* p=ret_rfsdevblk(devp,blknr);
    //把逻辑储存块中的数据复制到缓冲区中
    hal_memcpy(p,rdadr,FSYS_ALCBLKSZ);
    return DFCOKSTUS;
}
//获取超级块
rfssublk_t* get_superblk(device_t* devp)
{
    //分配4KB大小的缓冲区
    void* buf=new_buf(FSYS_ALCBLKSZ);
    //清零缓冲区
    hal_memset(buf,FSYS_ALCBLKSZ,0);
    //读取第0个逻辑储存块中的数据到缓冲区中，如果读取失败则释放缓冲区
    read_rfsdevblk(devp,buf,0);

    //返回超级块数据结构的地址，即缓冲区的首地址
    return (rfssublk_t*)buf;
}
//释放超级块
void del_superblk(device_t* devp,rfssublk_t* sbp)
{
    //回写超级块，因为超级块中的数据可能已经发生了改变，如果出错则死机
    write_rfsdevblk(devp,(void*)sbp,0);//释放先前分配的4KB大小的缓冲区
    del_buf((void*)sbp,FSYS_ALCBLKSZ);
    return;
}
//建立位图
bool_t create_bitmap(device_t* devp)
{
    bool_t rets=FALSE;
    //获取超级块，失败则返回FALSE
    rfssublk_t* sbp = get_superblk(devp);
    //分配4KB大小的缓冲区
    void* buf = new_buf(FSYS_ALCBLKSZ);
      //获取超级块中位图块的开始块号
    uint_t bitmapblk=sbp->rsb_bmpbks;
```

```
    //获取超级块中储存介质的逻辑储存块总数
    uint_t devmaxblk=sbp->rsb_fsysallblk;
    //如果逻辑储存块总数大于4096，就认为出错了
    if(devmaxblk>FSYS_ALCBLKSZ)
    {
        rets=FALSE;
        goto errlable;
    }
    //把缓冲区中每个字节都置成1
    hal_memset(buf,FSYS_ALCBLKSZ,1);
    u8_t* bitmap=(u8_t*)buf;
    //把缓冲区中的第3个字节到第devmaxblk个字节都置成0
    for(uint_t bi=2;bi<devmaxblk;bi++)
    {
        bitmap[bi]=0;
    }
    //把缓冲区中的数据写入到储存介质中的第bitmapblk个逻辑储存块中，即位图块中
    if(write_rfsdevblk(devp,buf,bitmapblk)==DFCERRSTUS){
        rets = FALSE;
        goto errlable;
    }
    //设置返回状态
    rets=TRUE;
errlable:
//释放超级块
    del_superblk(devp,sbp);
//释放缓冲区
    del_buf(buf,FSYS_ALCBLKSZ);
    return rets;
}
```

这里为什么又多了几个辅助函数呢？这是因为，位图块的块号和储存介质的逻辑储存块总数，都保存在超级块中，所以要实现获取、释放超级块的函数，还需要一个读取逻辑储存块的函数，写入逻辑储存块的函数前面已经写过了。

因为第 0 块是超级块，第 1 块是位图块本身，所以代码从缓冲区中的第 3 个字节开始清零，一直到 devmaxblk 个字节，devmaxblk 就是储存介质的逻辑储存块总数。缓冲区中有 4096 个字节，但 devmaxblk 肯定是小于 4096 的，所以 devmaxblk 后面的字节全部为 1，这样就不会影响到后面分配逻辑储存块代码的正确性了。

最后，我们把这个缓冲区中的数据写入到储存介质中的第 bitmapblk 个逻辑储存块中，就完成了位图的建立。

建立好了管理逻辑储存块状态的位图，下面就去接着建立根目录吧！

建立根目录

一切目录和文件都是存放在根目录下的，查询目录和文件也是从这里开始的，所以文件系统创建的最后一步就是创建根目录。

根目录也是一种文件，所以要为其分配相应的逻辑储存块，因为根目录下的文件和目录对应的 rfsdir_t 结构，就是保存在这个逻辑储存块中的。

因为根目录是文件，所以要在这个逻辑储存块的首个 512 字节空间中建立 fimgrhd_t 结构，即文件管理头数据结构。最后，我们要把这个逻辑储存块的块号，储存在超级块中的 rfsdir_t 结构中，同时修改该 rfsdir_t 结构中的文件名称为“/”。

要达到上述功能要求，就需要操作文件系统的超级块和位图，所以我们要先写好这些辅助功能函数，实现获取 / 释放位图块的代码如下所示。

复制代码

```
//获取位图块
u8_t* get_bitmapblk(device_t* devp)
{
    //获取超级块
    rfssublk_t* sbp = get_superblk(devp);
    //分配4KB大小的缓冲区
    void* buf = new_buf(FSYS_ALCBLKSZ);
    //缓冲区清零
    hal_memset(buf, FSYS_ALCBLKSZ, 0);
    //读取sbp->rsb_bmpbks块（位图块），到缓冲区中
    read_rfsdevblk(devp, buf, sbp->rsb_bmpbks)
    //释放超级块
    del_superblk(devp, sbp);
    //返回缓冲区的首地址
    return (u8_t*)buf;
}
//释放位图块
void del_bitmapblk(device_t* devp,u8_t* bitmap)
{
    //获取超级块
    rfssublk_t* sbp = get_superblk(devp);
    //回写位图块，因为位图块中的数据可能已经发生改变
    write_rfsdevblk(devp, (void*)bitmap, sbp->rsb_bmpbks)
    //释放超级块和存放位图块的缓冲区
    del_superblk(devp, sbp);
    del_buf((void*)bitmap, FSYS_ALCBLKSZ);
    return;
}
```

获取 / 释放位图块非常简单，就是根据超级块中的位图块号，把储存设备中的位图数据块读取到缓冲区中，而释放位图块则需要把缓冲区的数据写入到储存设备对应的逻辑块中。获取 / 释放超级块的函数，我们建立位图时已经写好了。

建立根目录需要分配新的逻辑储存块，分配新的逻辑储存块其实就是扫描位图数据，从中找出一个空闲的逻辑储存块，下面我们来写代码实现这个函数，如下所示。

复制代码

```
//分配新的空闲逻辑储存块
uint_t rfs_new_blk(device_t* devp)
{
    uint_t retblk=0;
    //获取位图块
    u8_t* bitmap = get_bitmapblk(devp);
    if(bitmap == NULL)
    {
        return 0;
    }
    for(uint_t blknr = 2; blknr < FSYS_ALCBLKSZ; blknr++)
    {
        //找到一个为0的字节就置为1，并返回该字节对应的空闲块号
        if(bitmap[blknr] == 0)
        {
            bitmap[blknr] = 1;
            retblk = blknr;
            goto retl;
        }
    }
    //如果到这里就说明没有空闲块了，所以返回0
    retblk=0;
retl:
    //释放位图块
    del_bitmapblk(devp,bitmap);
    return retblk;
}
```

rfs_new_blk 函数会返回新分配的逻辑储存块号，如果没有空闲的逻辑储存块了，就会返回 0。下面我们就可以建立根目录了，代码如下。

复制代码

```
//建立根目录
bool_t create_rootdir(device_t* devp)
```

```
{
    bool_t rets = FALSE;
    //获取超级块
    rfssublk_t* sbp = get_superblk(devp);
    //分配4KB大小的缓冲区
    void* buf = new_buf(FSYS_ALCBLKSZ);
    //缓冲区清零
    hal_memset(buf,FSYS_ALCBLKSZ,0);
    //分配一个空闲的逻辑储存块
    uint_t blk = rfs_new_blk(devp);
    if(blk == 0) {
        rets = FALSE;
        goto errlable;
    }
    //设置超级块中的rfsdir_t结构中的名称为“/”
    sbp->rsb_rootdir.rdr_name[0] = '/';
    //设置超级块中的rfsdir_t结构中的类型为目录类型
    sbp->rsb_rootdir.rdr_type = RDR_DIR_TYPE;
    //设置超级块中的rfsdir_t结构中的块号为新分配的空闲逻辑储存块的块号
    sbp->rsb_rootdir.rdr_blknr = blk;
    fimgrhd_t* fmp = (fimgrhd_t*)buf;
    //初始化fimgrhd_t结构
    fimgrhd_t_init(fmp);
    //因为这是目录文件所以fimgrhd_t结构的类型设置为目录类型
    fmp->fmd_type = FMD_DIR_TYPE;
    //fimgrhd_t结构自身所在的块设置为新分配的空闲逻辑储存块
    fmp->fmd_sfblk = blk;
    //fimgrhd_t结构中正在写入的块设置为新分配的空闲逻辑储存块
    fmp->fmd_curfwritebk = blk;
    //fimgrhd_t结构中正在写入的块的偏移设置为512字节
    fmp->fmd_curfinwbkoff = 0x200;
    //设置文件数据占有块数组的第0个元素
    fmp->fmd_fleblk[0].fb_blkstart = blk;
    fmp->fmd_fleblk[0].fb_blknr = 1;
    //把缓冲区中的数据写入到新分配的空闲逻辑储存块中，其中包含已经设置好的       fimgrhd_t
    if(write_rfsdevblk(devp, buf, blk) == DFCERRSTUS) {
        rets = FALSE;
        goto errlable;
    }
    rets = TRUE;
errlable:
    //释放缓冲区
    del_buf(buf, FSYS_ALCBLKSZ);
errlable1:
    //释放超级块
    del_superblk(devp, sbp);
    return rets;
}
```

上述代码的注释已经很清楚了，虽然代码有点长，但总体流程还是挺清晰的。首先，分配一块新的逻辑储存块。接着，设置超级块中的 rfsdir_t 结构中的名称以及类型和块号。然后设置文件管理头，由于根目录是目录文件，所以文件管理头的类型为 FMD_DIR_TYPE，表示文件数据存放的是目录结构。最后，回写对应的逻辑储存块即可。

串联

建立超级块、建立位图、建立根目录的代码已经写好了。

现在我们来写一个 rfs_fmat 函数，把刚才这三个操作包装起来，调用它们完成文件系统格式化这一流程。顺便，我们还可以把 init_rfs 函数也实现了，让它调用 rfs_fmat 函数，随后 init_rfs 函数本身会在 rfs_entry 函数的最后被调用，代码如下所示。

复制代码

```
//rfs初始化
void init_rfs(device_t *devp)
{
    //格式化rfs
    rfs_fmat(devp);
    return;
}
//rfs格式化
void rfs_fmat(device_t *devp)
{
    //建立超级块
    if (create_superblk(devp) == FALSE)
    {
        hal_sysdie("create superblk err");
    }
    //建立位图
    if (create_bitmap(devp) == FALSE)
    {
        hal_sysdie("create bitmap err");
    }
    //建立根目录
    if (create_rootdir(devp) == FALSE)
    {
        hal_sysdie("create rootdir err");
    }
    return;
}
//rfs驱动程序入口
drvstus_t rfs_entry(driver_t *drvp, uint_t val, void *p)
{
    //……

```

```
    init_rfs(devp);//初始化rfs
    return DFCOKSTUS;
}
```

上述代码中，init_rfs 函数会在 rfs 驱动程序入口函数的最后被调用，到这里我们 rfs 文件系统的格式化操作就完成了，这是实现文件系统的重要一步。

测试文件系统

尽管我们的文件系统还有很多其它操作，如打开、关闭，读写文件，这些文件相关的操作我们放在下一节课中来实现。这里我们先对文件系统格式化的功能进行测试，确认一下我们的格式化代码没有问题，再进行下一步的开发。

测试文件系统超级块

之前我们文件系统格式化操作的第一步，就是建立文件系统的超级块。

所以我们首先来测试一下建立文件系统超级块的代码，测试方法非常简单，我们只要把超级块读取到一个缓冲区中，然后把其中一些重要的数据，打印出来看一看就知道了，我们写个函数完成这个功能，代码如下所示。

复制代码

```
//测试文件系统超级块
void test_rfs_superblk(device_t *devp)
{
    kprint("开始文件系统超级块测试\n");
    rfssublk_t *sbp = get_superblk(devp);
    kprint("文件系统标识:%d,版本:%d\n", sbp->rsb_mgic, sbp->rsb_vec);
    kprint("文件系统超级块占用的块数:%d,逻辑储存块大小:%d\n", sbp->rsb_sblksz, sbp->
    kprint("文件系统位图块号:%d,文件系统整个逻辑储存块数:%d\n", sbp->rsb_bmpbks, sbp-
    kprint("文件系统根目录块号:%d 类型:%d\n", sbp->rsb_rootdir.rdr_blknr, sbp->rsb
    kprint("文件系统根目录名称:%s\n", sbp->rsb_rootdir.rdr_name);
    del_superblk(devp, sbp);
    hal_sysdie("结束文件系统超级块测试");//死机用于观察测试结果
    return;
}
//rfs驱动程序入口
drvstus_t rfs_entry(driver_t *drvp, uint_t val, void *p)
{
    init_rfs(devp);//初始化rfs
    test_rfs_superblk(devp);//测试文件系统超级块
    return DFCOKSTUS;
}
```

测试代码我们已经写好了，下面我们打开终端，切换到 Cosmos 目录下执行 make vboxtest，Cosmos 加载 rfs 驱动程序运行后的结果，如下所示。

文件系统超级块测试.

上图中我们可以看到，文件系统的标识、版本和最初定义的是相同的，逻辑储存块的大小为 4KB。位图占用的是第 1 个逻辑储存块，因为第 0 个逻辑储存块被超级块占用了。

同时，我们还可以看到储存设备上共有 1024 个逻辑储存块，根目录文件的逻辑储存块为第 2 块，名称为“/”，这些正确的数据证明了建立超级块的代码是没有问题的。

测试文件系统位图

测试完了文件系统超级块，我们接着来测试文件系统位图。测试方法很简单，先读取位图块到一个缓冲区中，然后循环扫描这个缓冲区，看看里面有多少个为 0 的字节，即表明储存介质上有多少个空闲的逻辑储存块。

我们一起来写好这个测试函数，代码如下所示。

复制代码

```
void test_rfs_bitmap(device_t *devp)
{
    kprint("开始文件系统位图测试\n");
    void *buf = new_buf(FSYS_ALCBLKSZ);
    hal_memset(buf, 0, FSYS_ALCBLKSZ);
    read_rfsdevblk(devp, buf, 1)//读取位图块
    u8_t *bmp = (u8_t *)buf;
    uint_t b = 0;
    //扫描位图块
    for (uint_t i = 0; i < FSYS_ALCBLKSZ; i++)
    {
        if (bmp[i] == 0)
        {
            b++;//记录空闲块
        }
    }
    kprint("文件系统空闲块数:%d\n", b);
    del_buf(buf, FSYS_ALCBLKSZ);
    hal_sysdie("结束文件系统位图测试\n");//死机用于观察测试结果
    return;
}
```

test_rfs_bitmap 函数我们已经写好了，别忘了**在 rfs_entry 函数的末尾调用它**，随后我们在终端下执行 make vboxtest，就可以看到 Cosmos 加载 rfs 驱动程序运行后的结果，如下所示。

文件系统位图测试

上图中的空闲块数为 1021，表示储存介质上已经分配了 3 块逻辑储存块了。这就证明了我们建立文件系统位图的代码是没有问题的。

测试文件系统根目录

最后我们来测试文件系统的根目录文件建立的对不对，测试方法就是先得到根目录文件的 rfsdir_t 结构，然后读取其中指向的逻辑储存块到缓冲区中，最后把它们的数据打印出来。

这个函数很简单，我们来写好它，代码如下。

复制代码

```
1 void test_rfs_rootdir(device_t *devp)
2 {
3     kprint("开始文件系统根目录测试\n");
4
```

```
    rfsdir_t *dr = get_rootdir(devp);
    void *buf = new_buf(FSYS_ALCBLKSZ);
    hal_memset(buf, 0, FSYS_ALCBLKSZ);
    read_rfsdevblk(devp, buf, dr->rdr_blknr)
    fimgrhd_t *fmp = (fimgrhd_t *)buf;
    kprint("文件管理头类型:%d 文件数据大小:%d 文件在开始块中偏移:%d 文件在结束块中的偏移:
            fmp->fmd_type, fmp->fmd_filesz, fmp->fmd_fileifstbkoff, fmp->fmd_f
    kprint("文件第一组开始块号:%d 块数:%d\n", fmp->fmd_fleblk[0].fb_blkstart, fmp-
    del_buf(buf, FSYS_ALCBLKSZ);
    del_rootdir(devp, dr);
    hal_sysdie("结束文件系统根目录测试\n");//死机用于观察测试结果
    return;
}
```

test_rfs_rootdir 函数同样要在 rfs_entry 函数的末尾调用，然后我们在终端下执行 make vboxtest，就可以看到 cosmos 加载 rfs 驱动程序运行后的结果了。

文件系统根目录测试

从上图我们可以看到，根目录文件的类型为目录文件类型。因为根目录文件才刚建立，所以文件大小为 0，文件数据的存放位置从文件占用的第 1 块逻辑储存块的 512 字节处开始。因为第 0、1 块逻辑储存块被超级块和位图块占用了，所以根目录文件占用的逻辑储存块，就是第 2 块逻辑储存块，只占用了 1 块。

好了，上面一系列的测试结果，表明我们的文件系统格式化的代码正确无误，文件系统格式化操作的内容我们就告一段落了

重点回顾

今天的课程就到这里了，今天我们继续推进了文件系统的进度，实现了文件系统的格式化操作，我来为你把今天的课程重点梳理一下。

首先实现了文件系统设备驱动程序框架，这是因为我们之前的架构设计，把文件系统作为 Cosmos 系统下的一个设备，这有利于**扩展不同的文件系统。**

然后我们实现了文件系统格式化操作，包括建立文件系统超级块、位图、根目录操作，并且将它们串联在一起完成文件系统格式化。

最后是对文件系统测试，我们通过打印出文件系统超级块、位图还有根目录的相关数据来验证，最终确认了我们文件系统格式化操作的代码是正确的。

虽然我们实现了文件系统的格式化，也对其进行了测试，但是我们的文件系统还是不能存放文件，因为我们还没有实现操作文件相关的功能，下一节课我们继续探索。

思考题

请问，建立文件系统的超级块、位图、根目录的三大函数的调用顺序可以随意调换吗，原因是什么？

欢迎你在留言区记录你的疑问或者收获，积极输出有利于你深入理解这节课的内容。同时，也欢迎你把这节课转给身边的同事、朋友。

好，我是 LMOS，我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 32 | 仓库结构：如何组织文件?

下一篇 34 | 仓库管理：如何实现文件的六大基本操作？

精选留言 (2)

写留言

青玉白露
2021-08-01

当然不行，从代码上可以看出来，这三者是相互依赖的，位图需要超级块，建立根目录需要位图来获取空闲的地址，差了哪一步都不行！

展开

pedro
2021-07-23

不能，位图初始化需要操作超级块，根目录初始化需要操作超级块和位图

作者回复: 铁子6666666